Développement formel par composants : assemblage et vérification à l'aide de B

Arnaud Lanoix, Samuel Colin, Jeanine Souquières

Développement formel par composants : assemblage et vérification à l'aide de B

Arnaud Lanoix* — **Samuel Colin*** — **Jeanine Souquières***

* *LORIA – Nancy Université*
Campus Scientifique, BP 239
F-54506 Vandœuvre lès Nancy cedex
{Firstname.Lastname}@loria.fr

RÉSUMÉ. *Dans une approche composants pour le développement de logiciels, les composants sont considérés comme des boîtes noires. Une application consiste en un assemblage de composants qui communiquent via leurs interfaces. Une description formelle de ces interfaces est nécessaire pour s'assurer de leur compatibilité. En général, les interfaces ne sont pas directement compatibles et un adaptateur doit être introduit. Nous proposons des schémas pour assembler des composants de manière systématique et vérifier leur interopérabilité; ces schémas sont définis à l'aide de concepts issus de la méthode B. L'assemblage est un raffinement des interfaces requises qui inclut les interfaces fournies; la correction du processus est validée par les obligations de preuves usuelles.*

ABSTRACT. *In a software component-based development approach, components are considered as black boxes. A component-based application consists of assembled components which communicate by the means of their interfaces. A formal description of these interfaces is therefore required to check their compatibility. In general, interfaces are not directly compatible and an adapter is required to bridge the gap. We propose patterns to assemble components in a systematic manner and verify their interoperability; these patterns are defined using B concepts. The assembly is a refinement of the required interfaces, including the provided interfaces; the process correctness is validated by the usual proof obligations.*

MOTS-CLÉS : *composant, adaptateur, assemblage, interface, vérification, construction sûre, raffinement.*

KEYWORDS: *component, adapter, assembly, interface, verification, thrustworthy development, refinement*

1re soumission à *Technique et Science Informatiques*, le 18 octobre 2007

1. Introduction

L'approche conception de systèmes par assemblage de composants est une approche de développement intéressante et de plus en plus adoptée (Szyperski, 1999). Une application à composants consiste en une composition de composants : des composants logiciels "boîte noire" développés par ailleurs sont assemblés les uns avec les autres pour produire le système complet. Le processus d'assemblage sous-jacent est similaire à celui des méthodes de construction et de réutilisation développées dans d'autres disciplines comme le génie mécanique ou le génie électrique.

Les composants sont assemblés via leurs interfaces. Une interface *fournie* par un composant peut être connectée avec une interface *requise* d'un autre composant si la première offre toutes les fonctionnalités permettant d'implanter la seconde : les composants doivent être connectés de manière appropriée. Afin de garantir cette *interopérabilité* entre composants, nous considérons chaque connexion entre interfaces fournie et requise de l'architecture et montrons que les interfaces sont compatibles. Une description appropriée des interfaces est primordiale si l'on veut vérifier que l'assemblage est correct.

Il est bien connu que la correction d'une connexion peut s'exprimer en termes de raffinement : l'interface fournie doit raffiner l'interface requise. La spécification formelle des interfaces et la preuve de leur interopérabilité en utilisant la méthode formelle B a été étudiée dans (Chouali *et al.*, 2006; Hatebur *et al.*, 2006). Grâce à B, nous prouvons que le modèle de l'interface fournie est un *raffinement* correct de l'interface requise ; en d'autres termes, nous prouvons que l'interface fournie correspond à une implantation correcte de l'interface requise et par conséquent, que les composants peuvent être connectés.

Dans une approche de réutilisation, ceci est insuffisant : les composants existants ont rarement des interfaces fournies qui raffinent directement l'interface requise du composant auquel on veut le connecter. Il faut bien souvent intercaler un adaptateur (ou médiateur) entre les deux composants pour les rendre compatibles ou bien encore développer un nouveau composant par assemblage de plusieurs composants existants pour répondre au besoin. Une première étude de la construction des adaptateurs et de leur preuve est décrite dans (Mouakher *et al.*, 2006). Dans cet article, nous proposons une approche systématique de développement formel par composants basée sur des schémas d'assemblages UML et B.

L'article est structuré de la manière suivante. La section 2 présente notre approche et l'utilisation de la méthode B. La section 3 présente différents types d'assemblage et pour chacun d'eux, un schéma d'architecture et son squelette en B permettant d'exprimer et de vérifier la correction de l'assemblage. Le problème particulier de la mise en correspondance de modèles de données est abordé dans la section 4. La section 5 illustre notre démarche avec le développement d'une étude de cas du contrôle d'accès à un bâtiment. Des travaux connexes sont discutés dans la section 6 et une conclusion avec des perspectives d'évolution termine cet article.

2. Description de l'approche et utilisation de B

Dans notre approche composants, l'architecture du système est modélisée à l'aide de différents diagrammes UML 2.0 (OMG, 2005) :

– les diagrammes de structure composite pour exprimer l'architecture globale du système en termes des composants et des interfaces à connecter ;

– les diagrammes de classes pour exprimer les modèles de données et les signatures des méthodes des interfaces ;

– les diagrammes de séquences pour exprimer les interactions possibles entre composants et décrire des protocoles d'usage complexe.

Le comportement autorisé ou attendu des interfaces est décrit à l'aide de modèles B.

2.1. *B et son utilisation dans notre approche*

B (Abrial, 1996) est une méthode formelle basée sur la théorie des ensembles, permettant un développement incrémental grâce au raffinement. Un développement commence avec la définition d'une spécification abstraite qui est ensuite raffinée pas à pas jusqu'à l'obtention d'une implantation. Un modèle B est composé de VARIABLES, d'un INVARIANT qui décrit les propriétés de ces variables et d'OPERATIONS qui définissent les possibles évolutions de ces variables. Un modèle B est correct s'il a été vérifié que les opérations préservent l'invariant. Un exemple de modèle B se trouve figure 12.

La méthode B a été appliquée avec succès dans le développement d'applications réelles complexes, comme le projet METEOR (Behm *et al.*, 1999) ou le métro Val (Badeau *et al.*, 2005). Elle s'appuie sur des outils robustes. Des obligations de preuves pour la consistance des invariants et la préservation du raffinement sont générées automatiquement par les outils (ste, 1998; Clearsy, 2004).

Dans notre approche, les interfaces des composants sont annotées de modèles B pour exprimer le comportement implanté (dans le cas d'une interface fournie) ou attendu (dans le cas d'une interface requise) par l'interface. Deux notions clés de la méthode B sont utilisées :

– le raffinement qui permet un développement incrémental avec préservation de la correction à chaque étape du développement,

– les mécanismes de composition, permettant un développement modulaire et la vérification de la correction lors de l'appel d'opérations.

NOTE. — Dans un processus de développement intégré, les modèles B pourraient être obtenus en appliquant des règles systématiques de transformation de UML vers B (Meyer *et al.*, 1999; Ledang *et al.*, 2001).
De plus, l'état actuel du développement de la plate-forme Rodin (RODIN, 2007) pour le B événementiel ne propose pas encore de mécanismes de (dé)composition, ce qui

nous empêche d'évoluer vers cette nouvelle plate-forme.

2.2. *Interopérabilité entre composants*

Deux composants peuvent être connectés via leurs interfaces respectives si ces interfaces sont *compatibles*. On dit qu'ils sont *interopérables*. En d'autres termes, l'interface fournie implante les fonctionnalités nécessaires à l'interface requise. Nous exprimons la correction de la connexion en termes de raffinement (Chouali *et al.*, 2006).

Définition. Soient deux composants OTS1[1] et OTS2 tels que OTS1 *requiert* une interface RI_ots1 et OTS2 *fournit* une interface PI_ots2. Ils sont interopérables si et seulement si le modèle B de PI_ots2 est un *raffinement* de celui de RI_ots1.

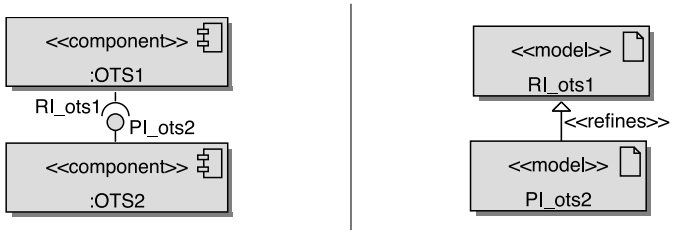


Figure 1 – Interopérabilité entre OTS1 et OTS2

REMARQUE. — Notons que l'interface fournie PI_ots2 peut offrir plus de fonctionnalités que n'en nécessite l'interface requise RI_ots1.

Lors d'un développement par composants, les problèmes suivants doivent être étudiés : vérifier que deux composants sont *interopérables*, sinon développer un adaptateur ; développer un nouveau composant à partir de composants existants.

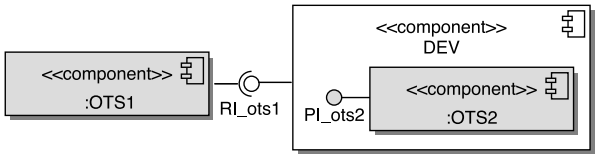


Figure 2 – Adaptation vs. développement d'un nouveau composant

Développer un adaptateur qui assure la bonne connexion entre un composant OTS1 et un composant OTS2, revient à développer un nouveau composant DEV qui fournira l'interface RI_ots1 en utilisant le composant OTS2 via son interface fournie PI_ots2, comme illustré figure 2.

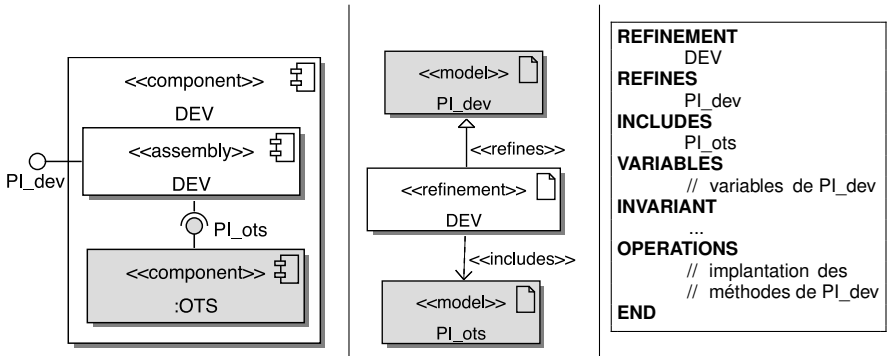
1. Nous nommons les composants existants OTS pour "Off-The-Shelf".

3. Différents assemblages de composants

Dans une approche de réutilisation, les composants existants ont rarement des interfaces directement compatibles. Un "nouveau" composant est nécessaire pour les rendre compatibles. Le développement de ce composant peut être plus ou moins complexe, en fonction du nombre et du type de composants existants à assembler. Nous étudions différents cas d'architecture et proposons des schémas pour assembler un ou plusieurs composants et vérifier la correction de l'assemblage.

3.1. *Cas de base : une seule interface dans l'assemblage*

Examinons la construction d'un composant DEV qui doit implanter une interface PI_dev. Il utilisera un composant existant OTS via son interface fournie PI_ots. Cette construction correspond à un *assemblage*, qui exprime comment les attributs et les méthodes de l'interface PI_dev sont implantés à l'aide ceux de PI_ots.



(a) Architecture UML (b) Architecture B (c) Squelette B de DEV

Figure 3 – Cas de base

1) chaque attribut de PI_dev est exprimé en termes des attributs de PI_ots ;

2) chaque méthode de PI_dev est exprimée par une combinaison d'appels aux méthodes pertinentes de PI_ots.

DEV est défini à l'aide d'un raffinement B, permettant de prouver la correction de l'assemblage. Nous proposons figure 3 un schéma d'assemblage illustrant l'architecture UML, les relations entre modèles B, ainsi qu'un squelette pour le modèle B de DEV. Les clauses VARIABLES, INVARIANT et OPERATIONS de ce squelette sont à compléter en accord avec les règles *1)* et *2)* énoncées ci-dessus.

La vérification des obligations de preuve assure que le modèle B de DEV

– *raffine* le modèle B associé à l'interface PI_dev (preuve de raffinement)

– en *incluant* correctement le modèle B associé à l'interface PI_ots (preuve d'inclusion)

c.à.d. que l'assemblage est correct au sens ou il réalise les besoins exprimés par PI_dev.

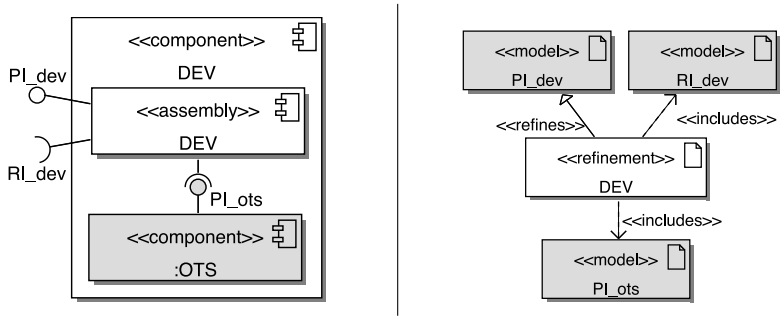


Figure 4 – Cas de base étendu

Il est possible que DEV requière une interface RI_dev. Cette interface exprime des besoins qu'il restera à implanter pour utiliser DEV. Néanmoins, DEV peut utiliser les attributs et les méthodes fournis par RI_dev (au même titre que ceux fourni par PI_ots) pour implanter PI_dev. Dans ce cas, le schéma d'assemblage devient celui proposé figure 4.

3.2. *Cas de deux interfaces dans l'assemblage*

Considérons maintenant le cas où le composant à utiliser requiert aussi une interface RI_ots pour implanter correctement les fonctionnalités fournies par PI_ots. Dans ce cas, DEV devra implanter l'interface RI_ots (pour répondre à OTS) en plus de l'interface PI_dev.

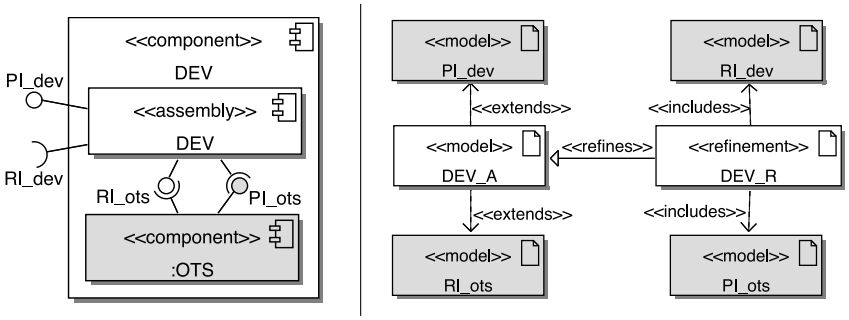


Figure 5 – Deux interfaces dans l'assemblage

La construction de DEV nécessite deux étapes de développement comme illustré figure 5 :

– l'introduction d'un modèle abstrait DEV_A qui *étend* les interfaces à implanter PI_dev et RI_ots pour les regrouper dans un seul modèle B[2] ;

– la définition d'un modèle DEV_R qui exprime l'assemblage. Il raffine DEV_A en incluant les modèles B des interfaces à utiliser, ici RI_dev et PI_ots, afin d'assurer que toutes les fonctionnalités à implanter le sont de manière correcte.

3.3. *Cas général : assemblage de plusieurs composants*

Nous généralisons notre démarche à l'assemblage de plusieurs composants qui fournissent et/ou requièrent des interfaces particulières afin de construire un nouveau composant. DEV doit réaliser l'ensemble des interfaces requises des composants à l'aide de leurs interfaces fournies. La figure 6 donne le schéma d'assemblage dans le cas de deux composants OTS1 et OTS2.

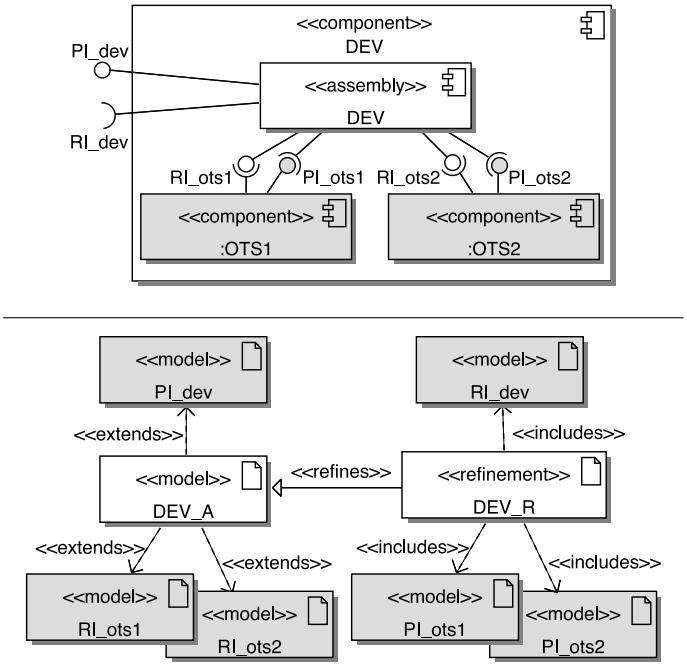


Figure 6 – Assemblage de plusieurs composants

Comme plusieurs composants sont en jeu, le protocole d'appels aux méthodes des interfaces fournies peut être complexe. Il peut être utile de commencer par l'exprimer à l'aide de diagrammes de séquences UML 2.0. À chaque méthode de DEV, on

2. Ce modèle intermédiaire est nécessaire parce que le B classique n'autorise le raffinement que d'un seul modèle à la fois.

associe l'enchaînement des appels de méthodes nécessaires des différentes interfaces permettant de la réaliser. L'écriture en B des différents diagrammes de séquences est facilitée et pourrait s'automatiser. La preuve de raffinement du modèle B permet de s'assurer que les appels aux méthodes fournies sont valides (preuves d'inclusion) et le raffinement en lui-même assure que l'implantation est correcte.
REMARQUE. — B propose un mécanisme de renommage associé à la clause INCLUDES qui permet d'utiliser, dans un assemblage de composants, plusieurs instances d'un même composant via des interfaces fournies identiques.

3.4. *Raffinement versus implantation en B*

Il est à noter que le REFINEMENT pourrait être une IMPLANTATION. L'avantage principal est de conclure définitivement l'adaptation de manière claire, en proposant un modèle réellement *implanté* en termes des composants utilisés. Les inconvénients sont en revanche plus nombreux :

– L'adaptation est définitive, ce qui signifie qu'il ne sera plus possible de préciser ou d'optimiser l'utilisation qui est faite des composants fournis. En ce sens, se limiter au REFINEMENT B est un avantage qui laisse ouvertes des optimisations futures.

– Il faut prendre en compte les contraintes de construction de B : les variables utilisées dans l'implantation doivent être déclarées comme concrètes, les modèles utilisés doivent être *importés* plutôt qu'inclus.

– Les constantes (fonctionnelles) introduites tout au long du raffinement doivent être valuées, i.e. il est nécessaire de leur donner une valeur concrète. Lorsque ces constantes se basent sur des ensembles abstraits, donc dont les éléments ne sont pas connus, la valuation n'est possible que si ces ensembles sont isomorphes à des sous-ensembles des entiers naturels. Cela signifie donc que les ensembles abstraits doivent pouvoir être remplacés par des sous-ensembles des entiers naturels.

Cela imposerait d'utiliser les hypothèses de bonne formation des ensembles abstraits pour construire les valeurs concrètes de ces constantes fonctionnelles. Cela est possible en B événementiel du fait de l'obligation d'explicitation des propriétés des ensembles porteurs introduits ; cela n'est pas possible en B classique car celles-ci sont implicites.

– Dans le cas où les éléments des ensembles de base sont énumérés (et donc explicités), les constantes fonctionnelles peuvent être décrites. En revanche, les outils de preuve utilisés ont montré un changement de comportement, en ce sens qu'ils tentaient d'utiliser les valeurs concrètes de ces ensembles et relations plutôt que des tactiques générales de raisonnement. Il en a résulté une preuve des raffinements plus difficile que dans le cas d'ensembles de bases abstraits.

En conclusion, ces inconvénients, qui pour la plupart se situent au niveau pratique plutôt que théorique, ont fait que nous nous sommes limités au raffinement.

4. Mise en correspondance des modèles de données

Il n'est pas toujours facile de réaliser chaque attribut à implanter en termes des attributs fournis, en particulier lorsque les modèles de données des interfaces PI_dev et PI_ots sont différents (figure 3(a)). Pour exprimer et vérifier cette correspondance, nous procédons par étapes successives, en utilisant le mécanisme de raffinement de B. DEV est développé par une série de raffinements successifs. Il est initialisé, au niveau le plus abstrait, par le modèle B de l'interface requise et se termine avec l'inclusion du modèle B de l'interface fournie. Le processus d'adaptation des modèles B se décompose en trois grandes étapes de raffinement. Une première étude a été menée dans (Colin *et al.*, 2007). Le schéma des modèles B de DEV est détaillé Figure 7.

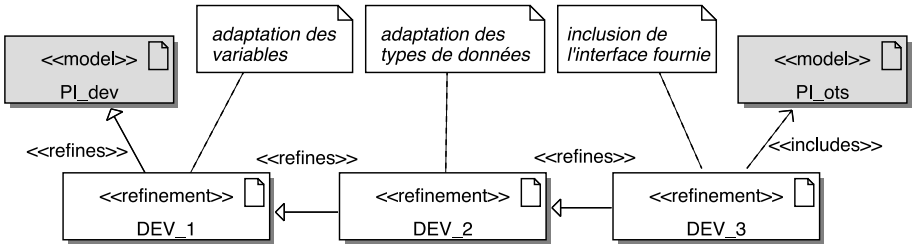


Figure 7 – Étapes pour la mise en correspondance de PI_dev avec PI_ots

1) Adaptation des variables. Cette étape prépare la mise en correspondance entre les attributs de PI_dev et ceux de PI_ots :

– de nouvelles variables, qui ré-expriment les variables de PI_dev sont introduites. Elles sont choisies afin de faciliter la mise en correspondance avec celles de PI_ots ;

– le corps de chaque opération de PI_dev est transformé pour prendre en compte ces nouvelles variables.

2) Adaptation des types de données. Cette étape correspond au transtypage des données :

– les variables introduites à l'étape précédente sont toujours exprimées en termes des types de données de PI_dev. Des fonctions de transtypage sont introduites afin de convertir les types de données de PI_dev vers ceux de PI_ots, et réciproquement. De nouvelles variables sont également introduites par l'application des fonctions de transtypage sur les variables introduites à l'étape précédente ;

– le corps de chaque opération de PI_dev est transformé pour tenir compte des modifications introduites sur les variables.

3) Inclusion de l'interface fournie. Les deux étapes précédentes ont servi à préparer cette dernière étape qui consiste à inclure le modèle B de l'interface fournie :

– puisque les variables de PI_dev ont été ré-exprimées et que les types de données ont été transformés, il est maintenant facile de mettre en correspondance les variables (modifiées) de PI_dev avec celles de PI_ots ;

– chaque opération de PI_dev est exprimée en termes d'appels aux opérations de PI_ots.

Ce processus de développement en trois étapes aide à construire l'assemblage, mais aussi à réaliser la preuve de l'adaptation. La preuve complète est facilitée par la décomposition en plusieurs étapes : il est plus facile de démontrer successivement chacune des étapes de l'adaptation plutôt que de démontrer l'ensemble des preuves en une seule étape. Il faut également souligner que les trois étapes ne sont pas toujours toutes nécessaires et qu'il est quelquefois plus facile de subdiviser l'une d'entre elles en plusieurs raffinements, toujours pour faciliter la preuve.

5. Etude de cas : le système de contrôle d'accès

Nous illustrons notre propos à l'aide de l'étude de cas du contrôle d'accès à un ensemble de bâtiments (AFADL'2000, 2000). L'objectif est de développer un système chargé de contrôler l'accès de personnes autorisées à un bâtiment donné d'un lieu de travail. Nous utilisons notre approche et les différents schémas d'assemblage proposés pour développer ce système à l'aide de composants préexistants.

5.1. *Le cahier des charges*

Le contrôle d'accès s'effectue sur la base de l'autorisation que chaque personne concernée possède. Cette autorisation doit lui permettre, sous le contrôle du système, d'entrer dans le bâtiment. Le nombre de personnes présentes dans le bâtiment doit être connu à tout instant.

Chaque personne autorisée dispose d'une carte d'accès avec un code. Des lecteurs de cartes sont installés à chaque entrée du bâtiment. À proximité de chaque lecteur se trouvent deux voyants, un rouge et un vert, chacun d'eux pouvant être allumé ou éteint. À chaque entrée et sortie du bâtiment se trouve un tourniquet normalement bloqué. Lorsqu'un tourniquet est débloqué par le système, le passage d'une personne est détecté par un capteur. Chaque tourniquet n'est affecté qu'à une seule tâche, entrer ou sortir. L'entrée obéit au protocole suivant :

– si la personne est autorisée à entrer dans le bâtiment (elle est toujours autorisée à sortir), le voyant vert s'allume et le tourniquet se débloque. Le cahier des charges originel fait état d'une contrainte de temps sur la durée de déblocage, contrainte que nous avons préféré abstraire en supposant qu'elle était gérée par le tourniquet lui-même. Dès que la personne franchit le tourniquet, le voyant vert s'éteint et le tourniquet se bloque immédiatement. Si la carte n'a pas été reprise par la personne au bout d'un certain laps de temps, elle est «avalée» par le lecteur ;

– si la personne n'est pas autorisée à entrer dans le bâtiment, le voyant rouge s'allume et le tourniquet reste bloqué. Ici encore, le retrait de la carte est soumis à une durée limite au-delà de laquelle la carte est «avalée» par le lecteur.

5.2. *Une architecture composants*

Dans une approche développement par composants, le système de contrôle d'accès peut être représenté par AccessControl, présenté figure 8. Les besoins auxquels ce système doit répondre sont exprimés à l'aide des interfaces suivantes :

– RI_Database permettra au contrôleur d'envoyer des requêtes à une base de données contenant les autorisations des usagers et des informations sur les personnes présentes dans les bâtiments ;

– RI_Entry permettra au contrôleur d'accès de commander le blocage/déblocage de l'entrée ; l'interface PI_Entry informera le contrôleur du passage d'une personne ;

– PI_Exit informera le contrôleur lorsqu'une personne sortira du bâtiment ;

– PI_Ident et RI_Ident devront proposer l'ensemble des fonctionnalités liées à l'identification par le contrôleur d'accès. Celui-ci commandera le système d'identification par le biais de l'interface RI_Ident et recevra des informations en retour via PI_Ident.

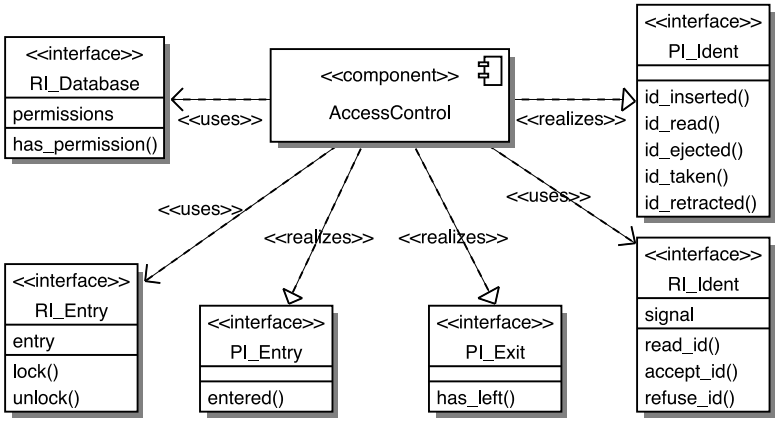


Figure 8 – AccessControl et ses interfaces

Un modèle B est associé à chacune de ces interfaces afin d'exprimer de manière précise les comportements et le protocole d'usage requis par le cahier des charges. À titre d'exemple, les modèles B de RI_Entry et RI_Database sont présentés figures 12 et 18.

5.2.1. *Composants existants*

Pour répondre aux besoins exprimés par AccessControl, nous disposons des composants présentés figure 9 :

– DBNetwork décrit un pilote permettant de connecter une base de données via son interface PI_DBNet. Le modèle B associé est proposé figure 18 ;

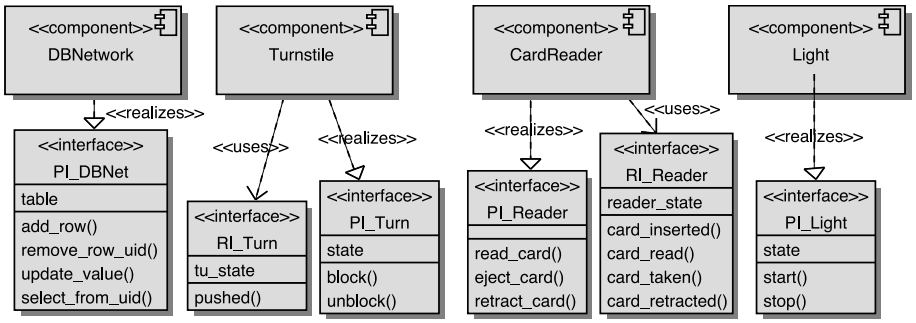


Figure 9 – Les composants DBNetwork, Turnstile, CardReader et Light

– Turnstile fournit un pilote chargé de commander un tourniquet. L'interface PI_Turn fournit des méthodes pour commander l'ouverture et la fermeture du tourniquet ; son modèle B est donné figure 12 ; RI_Turn propose une méthode pour informer du passage d'une personne ;

– CardReader fournit un pilote de périphérique à un lecteur de cartes. Ses deux interfaces PI_Reader et RI_Reader correspondent à l'interfaçage entre le lecteur de cartes et son environnement ;

– Light décrit le pilote de commande d'une lampe. Son interface PI_Light permet d'allumer et d'éteindre la lampe.

5.2.2. *Architecture globale*

L'architecture globale du système est décrite Figure 10 sous la forme d'un diagramme de structure composite UML. Pour répondre aux besoins exprimés par AccessControl en utilisant les composants existants présentés dans la section 5.2.1, il est nécessaire de développer des composants intermédiaires, qui assembleront (adapteront) ces composants pour répondre aux besoins : Entry, Exit, Identification et Database. Deux instances de Turnstile, notées turn_entry et turn_exit – pour l'entrée et la sortie du bâtiment – ainsi que deux instances de Light, notées red et green – pour les deux voyants rouge et vert – seront nécessaires.

Dans la suite du papier, nous détaillons le développement des composants Entry, Database et Identification.

5.3. *Le composant* Entry

Un adaptateur est nécessaire pour connecter AccessControl (via RI_Entry et PI_Entry) au composant Turnstile (via RI_Turn et PI_Turn). L'application du schéma correspondant au cas de deux interfaces dans l'assemblage donne l'architecture de ce composant, voir figure 11, dans laquelle :

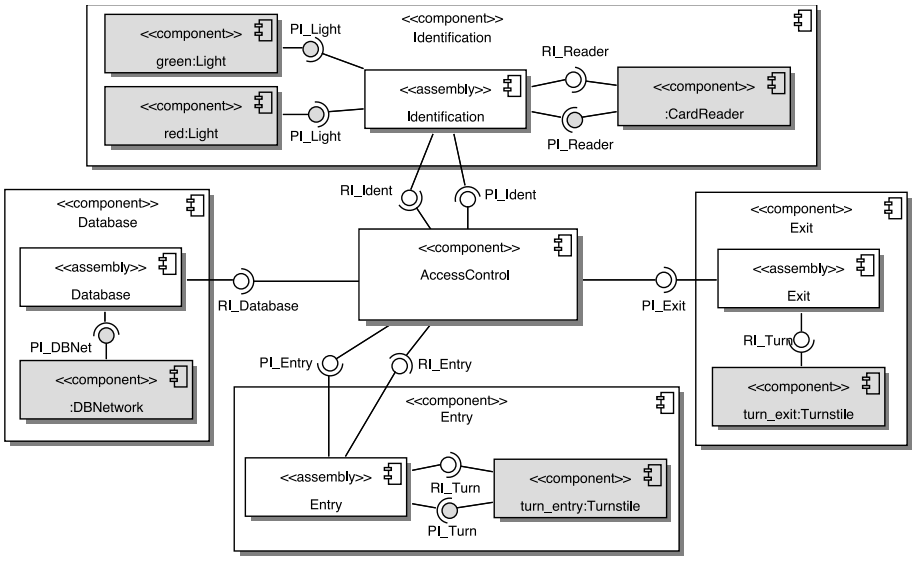


Figure 10 – Architecture globale du système de contrôle d'accès

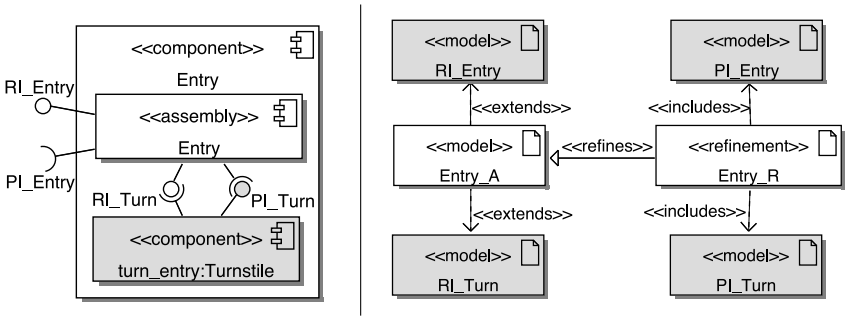


Figure 11 – Composant Entry

– Entry_A regroupe les interfaces à implanter ;

– Entry_R réalise l'assemblage.

Le modèle B raffiné Entry_R est complété pour exprimer l'implantation des éléments de RI_Entry et de RI_Turn en utilisant ceux de PI_Turn et PI_Entry : ici, il s'agit principalement d'exprimer des renommages, comme l'indique la figure 12. La preuve du raffinement de RI_Entry et de RI_Turn garantit que le modèle donné pour Entry est correct dans le sens où il réalise bien ce qui est attendu.

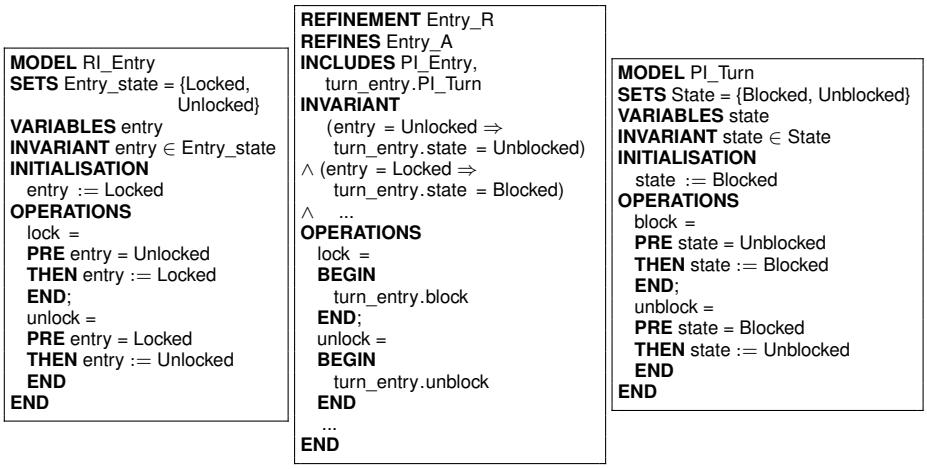


Figure 12 – Modèles B de RI_Entry, Entry_R et PI_Turn

5.4. *Le composant* Identification

Pour répondre aux besoins exprimés par RI_Ident et PI_Ident concernant le processus d'identification d'une personne, plusieurs composants devront être utilisés :

– CardReader via ses interfaces RI_Reader et PI_Reader et

– deux instances, green et red, du composant Light via son interface PI_Light.

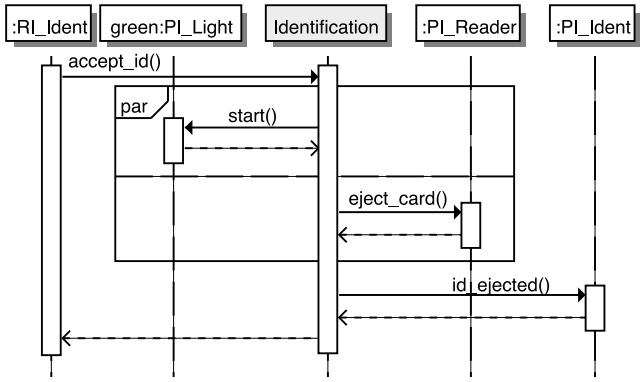


Figure 13 – Diagramme de séquences pour accept_id()

Des diagrammes de séquences peuvent être utilisés pour expliciter le protocole d'usage de Identification : à chaque méthode de chaque interface requise, on associe la réaction de l'assemblage, c.à.d. les appels aux méthodes nécessaires des interfaces fournies. Par exemple, la méthode accept_id() de RI_Ident correspond à une notifi-

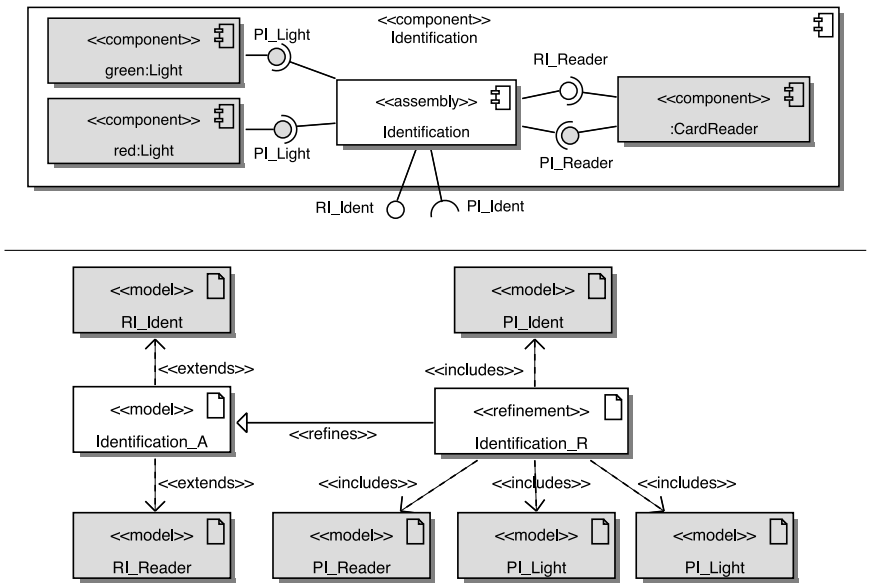


Figure 14 – Composant Identification

cation d'autorisation d'accès d'une personne par le système de contrôle d'accès. Son comportement attendu est décrit à l'aide du diagramme de séquences de la figure 13. L'adaptateur doit réagir en termes des interfaces fournies :

– en allumant la lampe verte (Green.start()) pour avertir l'usager de son autorisation d'accès tout en éjectant la carte (eject_card()), puis

– en notifiant au contrôleur d'accès l'éjection de la carte (id_ejected()).

Le composant Identification est obtenu par application du schéma d'assemblage de plusieurs composants, comme illustré figure 14. Le modèle B abstrait Identification_A regroupe les interfaces à implanter. Le modèle B de l'assemblage, Identification_R, raffine Identification_A afin d'assurer que l'assemblage est correct. Le code B présenté figure 15 complète le squelette du schéma :

– l'invariant établit un lien entre les variables à fournir reader_state et signal et les variables fournies par les interfaces PI_Ident, PI_Reader, green.PI_Light et red.PI_Light ;

– chaque méthode de RI_Reader et de RI_Ident est exprimée en termes d'appels aux méthodes correspondantes des modèles inclus. La méthode accept_id() correspond à une réécriture en B du diagramme de séquences donné figure 13. La méthode card_taken() correspond également à la réécriture du diagramme de séquences correspondant, donné figure 16.

```
REFINEMENT Identification_R
REFINES Identification_A
INCLUDES
  red.PI_Light, green.PI_Light,
  PI_Reader,
  PI_Ident
VARIABLES reader_state
INVARIANT
  (green.state = On ⇒
    red.state = Off) ∧
  (red.state = On ⇒
    green.state = Off) ∧
  (signal = No_signal ⇒
      (green.state = Off ∧
       red.state = Off))
```

```
OPERATIONS
  accept_id =
  BEGIN
    BEGIN
      green.start || eject_card
    END ;
    id_ejected
  END;
  ...
  card_taken =
  BEGIN
    id_taken ;
    SELECT red.state = On
      THEN red.stop
    WHEN green.state = On
      THEN green.stop
    END;
    reader_state := RWaiting
  END;
  ...
END
```

Figure 15 – Extraits de Identification_R

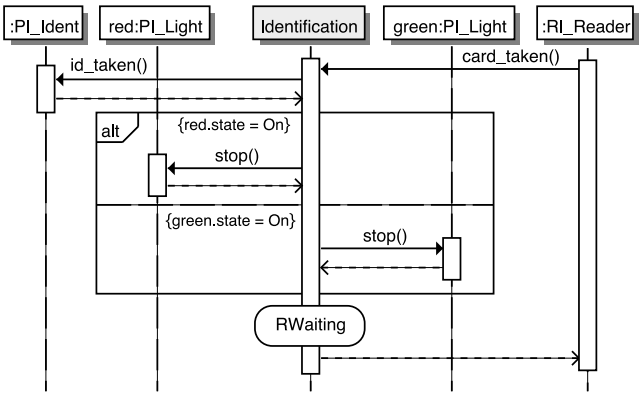


Figure 16 – Diagramme de séquences pour card_taken()

REMARQUE. — Les méthodes accept_id() et card_taken() illustrent les possibilités d'une traduction "automatique" d'un diagramme de séquences UML 2.0 vers une spécification B. Les concepts manipulés dans les diagrammes de séquences (séquences, parallèles et alternatives) sont également des concepts présents dans le langage B.

5.5. *Le composant* Database

Les interfaces RI_Database et PI_DBNet présentent des modèles de données

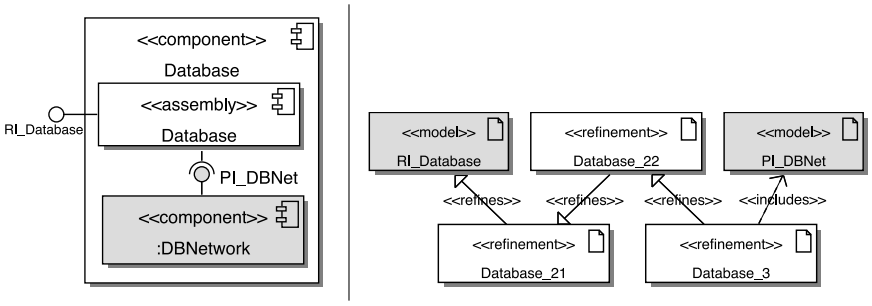


Figure 17 – Composant Database

différents :

– RI_Database permet d'obtenir les portes (bâtiments) autorisées pour une personne donnée ;

– PI_DBNet permet de mémoriser dans une base de données des couples (Uid, Value) d'entiers naturels.

L'adaptation est effectuée en appliquant le schéma de base lorsqu'une seule interface est en jeu dans l'assemblage avec la mise en correspondance des modèles de données. Les schémas de l'architecture UML et B du composant Database sont présentés figure 17. Plusieurs étapes de raffinement sont nécessaires pour développer le modèle B associé :

1) L'étape d'**adaptation des variables** n'est pas nécessaire puisque celles-ci peuvent être facilement mises en correspondance (voir figure 18) : les utilisateurs sont mis en correspondance avec le champ Uid de la base de données, et les portes avec le champ Value.

2) L'**adaptation des types de données** est décomposée en deux étapes afin de faciliter la preuve :

- dans Database_21, une fonction de transtypage user_cast est introduite afin de transformer le domaine de la relation permissions en le domaine les entiers naturels ; une nouvelle variable n_permissions est également introduite ;

- il s'agit maintenant de transformer le codomaine de n_permissions en le domaine des entiers naturels. Une fonction de transtypage door_cast, ainsi qu'une nouvelle variable nn_permissions, sont introduites dans Database_22.

3) La **dernière étape** consiste à associer les attributs Uid et Value de PI_DBNet à nn_permissions, c'est-à-dire à préciser les relations entre les structures de données. C'est également lors de cette étape que le corps des méthodes se réduit à l'appel des méthodes correspondantes du composant fourni (ici, has_permission appelle select_from_uid).

REMARQUE. — Les états des interfaces mises en œuvre restent disjoints, c.à.d. qu'il n'est pas possible de lier ces états, que ce soit au niveau des préconditions des mé-

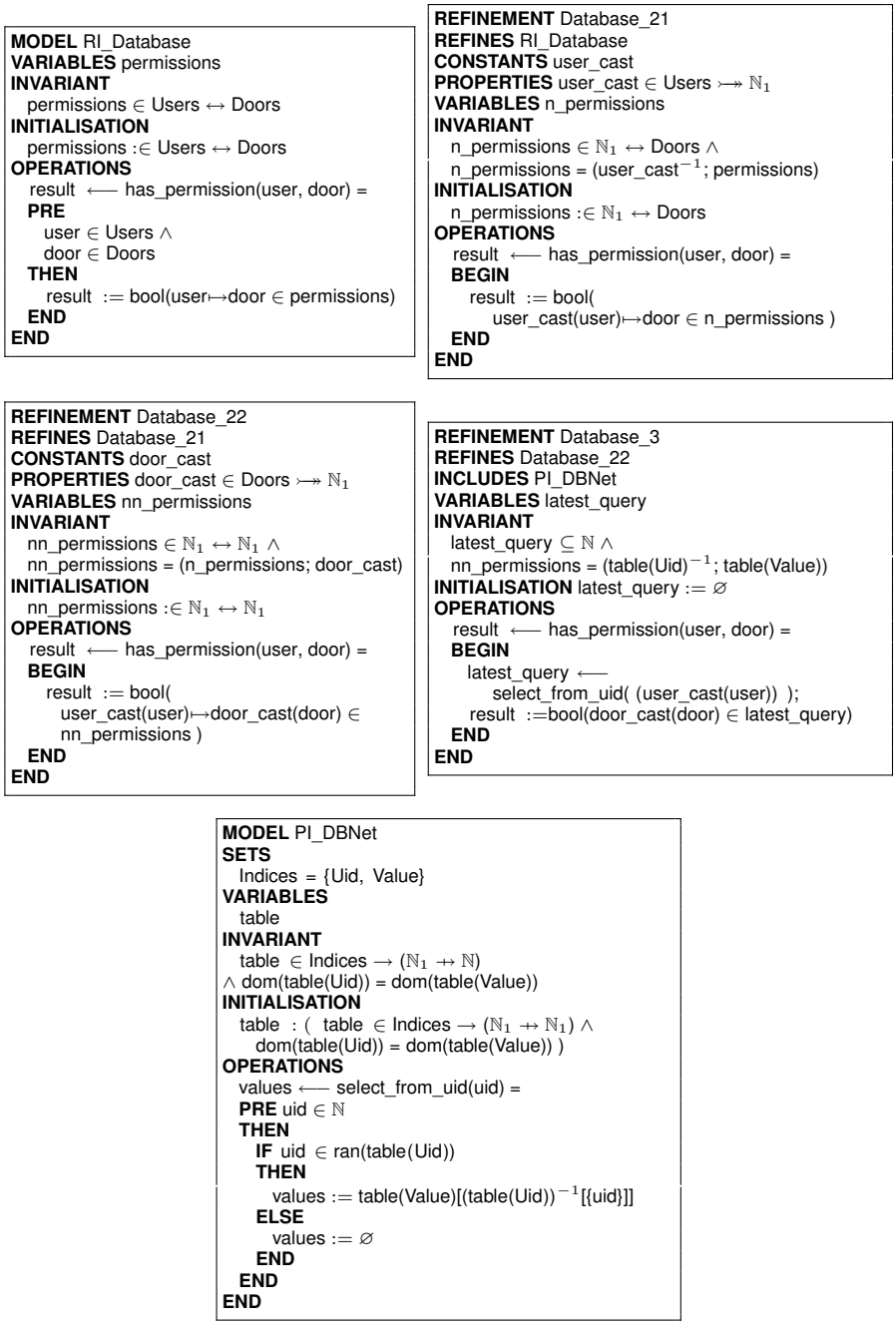


Figure 18 – Mise en correspondance des modèles de données

thodes ou des modifications effectuées, en utilisant les nouveaux états introduits dans le raffinement. Ce phénomène est induit par l'utilisation de composants indépendants. L'assemblage doit créer des liens qui n'existent pas : ceux-ci imposent un style de programmation défensif, traduit par l'utilisation dans notre exemple de gardes (clause SELECT) plutôt que de préconditions (style offensif).

Modèles B\OPs	évidentes	automatiques	interactives
Types	1	0	0
PI_Turn	5	0	0
RI_Turn	3	0	0
PI_Entry	3	0	0
RI_Entry	5	0	0
PI_Exit	3	0	0
Entry	**12**	**2**	**0**
Exit	**3**	**0**	**0**
PI_Light	5	0	0
PI_Reader	7	0	0
RI_Reader	9	0	0
PI_Ident	11	0	0
RI_Ident	7	0	0
Identification_A	**9**	**0**	**0**
Identification_R	**62**	**6**	**2**
PI_DBNet	12	10	4
RI_Database	3	0	0
Database_21	**6**	**2**	**2**
Database_22	**6**	**2**	**2**
Database_3	**5**	**8**	**2**
TOTAL	**177**	**30**	**12**

Tableau 1 – Obligations de preuves

Les différents assemblages présentés ainsi que les interfaces nécessaires ont tous été validés avec B4free. Cela nous permet d'assurer que les nouveaux composants développés par assemblage de composants existants sont corrects. Le détail des obligations de preuves (OPs) est donné dans le tableau 1.

6. État de l'art

Les travaux de recherche relatifs à l'adaptation de composants sont nombreux et la nécessité de disposer de mécanismes d'assemblage performants pour les réaliser a été reconnue dès les années 1990 (Brown *et al.*, 1996; Heineman *et al.*, 1999; Heisel *et al.*, 2002; Canal *et al.*, 2006).

Une des premières approches concernant la réutilisation de modules avec adaptation de leurs interfaces est celle proposée par Purtilo et Atlee (Purtilo *et al.*, 1991) : ils proposent un langage dédié, Nimble, où l'adaptation entre interfaces requises et fournies est effectuée par le développeur. Notre approche est assez voisine avec l'utilisation de UML et B comme langages, reposant sur des standards et des outils de vérification.

Des approches pragmatiques ont porté sur l'analyse des problèmes sous-jacents à l'adaptation de composants existants.Une définition formelle de l'interopérabilité et de l'adaptation de composants a été introduite dans (Yellin *et al.*, 1997). Dans ce cadre, la spécification du comportement d'un composant est décrite à l'aide de machines à états finis pour lesquelles il existe des techniques et des outils efficaces permettant la vérification de la compatibilité des protocoles.

Zaremski et Wing (Zaremski *et al.*, 1997) proposent une approche intéressante pour comparer deux composants logiciels, permettant de décider si un composant peut être remplacé par un autre. Ils utilisent les spécifications algébriques pour modéliser le comportement des composants et le prouveur Larch pour prouver la correspondance entre composants.

Reussner et Schmidt considèrent une certaine classe de problèmes dans le contexte des systèmes concurrents (Schmidt *et al.*, 2002; Reussner *et al.*, 2003). L'incompatibilité des protocoles est résolue par la génération d'adaptateurs en utilisant les interfaces décrites en termes de machines à états finis.

Les travaux présentés dans (Poizat *et al.*, 2006) proposent un processus de génération d'adaptateurs. De nombreux travaux actuels sont dédiés à l'adaptation dynamique (WCAT2006, 2006), qui va plus loin que notre approche : l'adaptation des composants s'effectue lors de l'exécution en recherchant le composant adapté (Mätzel *et al.*, 1997; Kniesel, 1999). Ces méthodes se basent sur l'hypothèse de l'existence de relations d'héritages (avec une possible transitivité) entre une interface fournie et une classe qu'on sait pouvoir utiliser. Elles sont fortement basées sur la notion de sous-typage dans un contexte de programmation objet, et donc sont moins flexibles en termes d'expressivité que notre approche, bien qu'elles apportent l'adaptation dans un contexte dynamique.

Ehrig & al (Ehrig *et al.*, 2004) présentent un cadre pour modéliser des architectures composants en utilisant des techniques formelles telles que les réseaux de Petri et CSP : les connexions entre interfaces requises et fournies sont représentées par des transformations de graphes utilisant des notions de composition, d'extension et de raffinement. Notre approche est similaire avec l'utilisation de B pour exprimer les transformations comme des raffinements entre interfaces requises et fournies.

Bracciali & al (Bracciali *et al.*, 2005) spécifient un adaptateur comme un ensemble de correspondances entre les méthodes et les paramètres des composants requis et fournis. Un adaptateur est formalisé par un ensemble de propriétés exprimées à l'aide du π-calcul.

La génération automatique d'adaptateurs est limitée à une certaine classe de problèmes car la vérification de l'interopérabilité repose sur la décidabilité de l'inclusion des composants. Dans notre approche, nous proposons des schémas pour construire et vérifier les adaptateurs, en fonction de différents cas de figures de l'architecture, sans aller jusqu'à leur génération automatique.

7. Conclusion

L'approche composants est un paradigme bien connu et utilisé dans le développement de logiciels, aussi bien dans le milieu académique que dans le milieu industriel. Dans cette approche, les composants sont considérés comme des boîtes noires décrites en termes de leur comportement visible et de leurs interfaces, qu'elles soient requises ou fournies. Ils sont assemblés via leurs interfaces. Il est bien connu que la correction d'une connexion peut s'exprimer en termes de raffinement : l'interface fournie doit raffiner l'interface requise.

Dans une approche de réutilisation de composants existants, les composants ont rarement des interfaces fournies qui raffinent directement l'interface requise du composant auquel on veut le connecter. Il est nécessaire d'introduire un adaptateur entre les composants pour les rendre compatibles. Un adaptateur est un programme qui définit comment les interfaces requises sont réalisées en termes des interfaces fournies : il exprime la correspondance entre variables, types et opérations.

Nous avons proposé une approche systématique de développement formel par composants basée sur des schémas d'assemblages UML et B. Ces schémas permettent d'exprimer la notion d'adaptateur et plus généralement le développement d'un nouveau composant par assemblage de composants préexistants , en fonction des différents cas de figures de l'architecture. Nous ne proposons pas de les générer automatiquement.

Grâce à l'utilisation de la méthode B, et de ses mécanismes d'assemblage et de raffinement pour modéliser les interfaces et les adaptateurs, nous obtenons la preuve de la correction de l'assemblage de composants. Le prouveur B garantit que cet as-

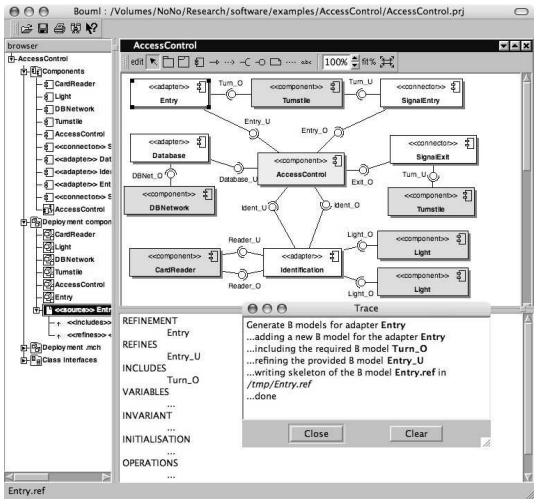


Figure 19 – BOUML pour générer un modèle B

semblage est une implantation correcte des fonctionnalités attendues en termes des composants existants.

Les points forts de notre approche sont :

– l'utilisation de notations simples et de haut niveau pour exprimer l'architecture du système et ses interfaces ;

– des schémas d'assemblage utilisant les mécanismes classiques de composition et de raffinement ;

– un guide pour développer incrémentalement de nouveau composants ;

– la preuve de la correction de l'assemblage des composants.

L'implantation d'un plugin pour BOUML[3] fondé sur les schémas de développement présentés dans cet article est en cours : la figure 19 montre la génération du squelette du modèle B correspondant à l'assemblage Entry.

L'extension de l'approche avec la prise en compte de propriétés de sécurité dans une architecture composants existante, sans modification de ses fonctionnalités de base (Lanoix *et al.*, 2007) est en cours d'étude. Ce travail doit également être complété par un outil d'aide à la détection des incompatibilités.

Remerciements

Ce travail a bénéficié d'une aide de l'Agence Nationale de la Recherche dans le cadre du projet TACOS[4], référence ANR-06-SETI-017.

8. Bibliographie

Abrial J.-R., *The B Book*, Cambridge University Press, 1996.

AFADL'2000, « Etude de cas : système de contrôle d'accès », *Journées AFADL, Approches formelles dans l'assistance au développement de logiciels*, 2000. actes LSR/IMAG.

Badeau F., Amelot A., « Using B as a High Level Programming Language in an Industrial Project : Roissy VAL », *ZB 2005 : Formal Specification and Development in Z and B, 4th International Conference of B and Z Users*, vol. 3455 of *LNCS*, Springer-Verlag, p. 334-354, 2005.

Behm P., Benoit P., Meynadier J., « METEOR : A Successful Application of B in a Large Project », *Integrated Formal Methods, IFM99*, vol. 1708 of *LNCS*, Springer Verlag, p. 369-387, 1999.

Bracciali A., Brogi A., Canal C., « A Formal Approach to Component Adaptation », *Journal of Systems and Software*, vol. 74, Elsevier Science Inc., p. 45-54, 2005.

3. http ://bouml.free.fr
4. http ://tacos.loria.fr

Brown A. W., Wallnan K. C., « Engineering of component-based systems », *Proceedings of the 2nd IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '96)*, IEEE Computer Society, p. 414, 1996.

Canal C., Murillo J. M., Poizat P., « Software Adaptation », *L'Objet*, vol. 12, n° 1, p. 9-31, 2006.

Chouali S., Heisel M., Souquières J., « Proving Component Interoperability with B Refinement », *Electronic Notes in Theoretical Computer Science (ENTCS)*, vol. 160, p. 157-172, 2006.

Clearsy, « B4free », , website, 2004. `http://www.b4free.com`.

Colin S., Lanoix A., Souquières J., « trustworthy interface compliancy : data model adaptation », *Formal Foundations of Embedded Software and Component-Based Software Architectures (FESCA), Satellite workshop of ETAPS*, March, 2007. 13 pages. To be published in Electronic Notes in Theoretical Computer Science (ENTCS).

Ehrig H., Padberg J., Braatz B., Klein M., Orejas F., Perez S., Pino E., « A Generic Framework for Connector Architectures based on Components and Transformation », *FESCA'04, satellite of ETAPS'04*, vol. 108 of *ENTCS*, p. 53-67, 2004.

Hatebur D., Heisel M., Souquières J., « A Method for Component-Based Software and System Development », *Proceedings of the 32nd Euromicro Conference on Software Engineering And Advanced Applications*, IEEE Computer Society, p. 72-80, 2006.

Heineman G., Ohlenbusch H., An Evaluation of Component Adaptation Techniques, Technical Report n° WPI-CS-TR-98-20, Department of Computer Science, Worcester Polytechnic Institute, February, 1999.

Heisel M., Santen T., Souquières J., « Toward a formal model of software components », *Proc. 4th International Conference on Formal Engineering Methods - ICFEM'02*, n° 2495 in *LNCS*, Springer-Verlag, p. 57-68, 2002.

Kniesel G., « Type-Safe Delegation for Run-Time Component Adaptation », *Lecture Notes in Computer Science*, vol. 1628, p. 351-366, 1999.

Lanoix A., Hatebur D., Heisel M., Souquières J., « Enhancing Dependability of Component-based Systems », *in* S. Verlag (ed.), *Reliable Software Technologies Ada-Europe 2007*, n° 4498 in *LNCS*, Springer Verlag, p. 41-54, 2007.

Ledang H., Souquières J., « Modeling class operations in B : application to UML behavioral diagrams », *ASE'2001 : 16th IEEE International Conference on Automated Software Engineering*, IEEE Computer Society, p. 289-296, 2001.

Meyer E., Souquières J., « A systematic approach to transform OMT diagrams to a B specification », *Proceedings of the Formal Method Conference*, n° 1708 in *LNCS*, Springer-Verlag, p. 875-895, 1999.

Mouakher I., Lanoix A., Souquières J., « Component Adaptation : Specification and Verification », *Proc. of the 11th Int. Workshop on Component Oriented Programming (WCOP'06), satellite workshop of ECOOP 2006*, p. 23-30, 2006.

Mätzel K.-U., Schnorf P., Dynamic Component Adaptation, Technical report, Ubilab laboratory, Union Bank of Switzerland, Zürich, Switzerland, June, 1997.

OMG, *UML Superstructure Specification*. 2005, version 2.0.

Poizat P., Salaün G., Tivoli M., « An Adaptation-based Approach to Incrementally Build Component Systems », *FACS'06*, Electronic Notes in Theoretical Computer Science, 2006. to appear.

Purtilo J., Atlee J., « Module Reuse by Interface Adaptation », *Software - Practice and Experience*, vol. 21, n° 6, p. 539-556, 1991.

Reussner R. H., Schmidt H. W., Poernomo I. H., « Reasoning on Software Architectures with Contractually Specified Components », *in* A. Cechich, M. Piattini, A. Vallecillo (eds), *Component-Based Software Quality : Methods and Techniques*, p. 287-325, 2003.

RODIN, « Rigorous Open Development Environment for Complex Systems », , `http://rodin-b-sharp.sourceforge.net`, Aug, 2007.

Schmidt H. W., Reussner R. H., « Generating Adapters fo Concurrent Component Protocol Synchronisation », *in* I. Crnkovic, S. Larsson, J. Stafford (eds), *Proceeding of the 5th IFIP International conference on Formal Methods for Open Object-based Distributed Systems*, p. 213-229, 2002.

ste, *Obligations de preuve : Manuel de référence, version 3.0*. 1998.

Szyperski C., *Component Software*, ACM Press, Addison-Wesley, 1999.

WCAT2006, « Coordination and Adaptation Techniques : Bridging the Gap Between Design and Implementation », *in* S. Becker, C. Canal, N. Diakov, J.-M. Murillo, P. Poizat, M. Tivoli (eds), *Proceedings of the Third International Workshop on Coordination and Adaptation Techniques for Software Entities*, 2006.

Yellin D. D. M., Strom R. E., « Protocol Specifications and Component Adaptors. », *ACM Transactions on Programming Languages and Systems*, vol. 19, n° 2, p. 292-333, 1997.

Zaremski A. M., Wing J. M., « Specification matching of software components », *ACM Transaction on Software Engeniering Methodolology*, vol. 6, n° 4, p. 333-369, 1997.